ARI Research Note 96-40

Graphical Representations and Causal Models in Intelligent Interactive Learning Environments

Brian J. Reiser
Princeton University

Research and Advanced Concepts Office
Michael Drillings, Acting Director

March 1996

19960607 165

DTIC QUALITY INSPECTED 3

United States Army
Research Institute for the Behavioral and Social Sciences

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED (from. . . to)
1996, March	Final	September 1990-August 1991

4. TITLE AND SUBTITLE	5a. CONTRACT OR GRANT NUMBER
Graphical Representations and Causal Models in Intelligent Learning Environments	MDA903-90-C-0123
	5b. PROGRAM ELEMENT NUMBER 0601102A
6. AUTHOR(S) Brian J. Reiser (Princeton University)	5c. PROJECT NUMBER B74F
	5d. TASK NUMBER 3901
	5e. WORK UNIT NUMBER C48
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Department of Psychology Princeton University Washington Road Princeton, NJ 08544-1010	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Research Institute for the Behavioral and Social Sciences ATTN: PERI-BR 5001 Eisenhower Avenue Alexandria, VA 22333-5600	10. MONITOR ACRONYM ARI
	11. MONITOR REPORT NUMBER Research Note 96-40

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

COR: George Lawton

14. ABSTRACT *(Maximum 200 words)*:

This research project investigates how an interactive learning environment can support students' learning and acquisition of mental models when acquiring a target cognitive skill. In this project, we have constructed GIL, an intelligent tutoring system for LISP programming, and have used GIL to conduct pedagogical experiments on skill acquisition. Progress in the current year includes extensions to GIL's graphical representation and model tracing capabilities. The experiments run this year include a study of how GIL's graphical representations facilitate learning for complex programming skills and how GIL enables students to engage in a more natural reasoning than the traditional text-based representation of the programming language.

15. SUBJECT TERMS

Problem solving Intelligent tutoring systems Skill acquisition LISP

SECURITY CLASSIFICATION OF			19. LIMITATION OF ABSTRACT	20. NUMBER OF PAGES	21. RESPONSIBLE PERSON (Name and Telephone Number)
16. REPORT	17. ABSTRACT	18. THIS PAGE			
Unclassified	Unclassified	Unclassified	Unlimited	27	

Contents

1 Introduction and Overview

This project investigates how an interactive learning environment can support students' learning and acquisition of mental models when acquiring a target cognitive skill. In particular, the research program explores how graphical representations and intelligent guidance can facilitate students' learning in a new domain.

Our approach is to construct an intelligent interactive learning environment, and to use the learning environment to conduct pedagogical experiments on skill acquisition. We have constructed an intelligent tutoring system called GIL (*Graphical Instruction in LISP*) that teaches students to program in LISP. GIL functions in both a guided tutoring mode and as a more open-ended exploratory learning environment. In its guided model tracing mode, GIL provides explanatory feedback in response to student errors or requests for hints by comparing the student's solution so far to the range of solutions suggested by its problem solver. In its exploratory mode, GIL provides support for students to articulate their hypotheses about how a program will behave, and tools for testing their hypotheses. GIL's graphical representations and causal explanations work together to help students build and capitalize on an effective mental model for programming.

In this report, we review our progress in the first year of our contract, 9-1-90 through 8-31-91. We will first discuss the empirical investigations of skill acquisition conducted using GIL, and then discuss our progress in extending GIL's model tracing and graphical interface components.

2 Studies of How Graphical Representations Facilitate Reasoning

2.1 Experiment One: GIL vs Text LISP

Our previous evaluations of GIL's effectiveness were based on the material covered in the first two sections of GIL's curriculum (Reiser, Ranney, Lovett, & Kimberg, 1989). We expect the benefits of GIL's graphical representation to provide even stronger advantages for students' learning as the curriculum progresses in difficulty. For example, in many cases students encounter difficulty mastering the use of conditional expressions and logical reasoning in programming, because they construct models that are at odds with the programming language's behavior (Reiser, Kimberg, Lovett, & Ranney, 1991). We expect that GIL's graphical representation will help students to understand more clearly the behavior of these programming constructs.

To test GIL's effectiveness for a more advanced curriculum, we compared two

groups of novice students learning LISP (Reiser, Beekelaar, Tyle, & Merrill, 1991). The curriculum covered in this experiment is shown in Table 1. The *text LISP* students learned LISP using a standard programming environment containing a simple editor and an interactive LISP interpreter. The GIL group learned to program using GIL. Both groups read the same textbook material and worked at their own pace. To ensure that any benefits were due to the support of the graphical interface, we used GIL without its model tracing feedback in the part of the curriculum under investigation (the lesson on conditionals Sections 5-8). In this study, both groups constructed programs using variables, constants, and the same 24 basic LISP functions.

We found that the GIL students mastered the material more quickly than the text LISP students. There was a large difference in time to solve the assigned problems — the GIL students solved these more advanced problems approximately 40% faster than students using text LISP.

There are several ways in which these problem solving advantages may arise. It is possible that students using GIL solve problems more quickly because the syntax of the graphical representation is simpler than the syntax of text LISP. The syntax of LISP, like that of any programming language, poses problems for novices. Indeed, more than half of the modifications of the text LISP students' programs were syntactic changes (i.e., additions or deletions of quotes or parentheses that do not change the intended sequence of operations of the program). In contrast, GIL representations contain few syntactic elements, relying instead on visual indications such as a line between two icons or the left/right order of two branches. Thus, GIL students rarely made syntactic changes to programs, making almost all of their changes to the algorithm. It is important to stress, however, that the benefits of GIL cannot be attributed solely to a simpler syntax. Even when syntactic changes are excluded from analysis, GIL students still required fewer modifications to solve the problems correctly. Thus, GIL students were better able to find and repair the problems in their programs than were the students working with text LISP.

Another indication of the greater difficulty experienced by text LISP students is the presence in the solutions of irrelevant components, often called *dead code*. Dead code is a portion of a program that can never have an effect on its output. For example, in the expression *(and (listp variable))*, the expression would return the same value if the *and* function call were excluded, so that function is dead code. Interestingly, more than three times as many solutions from the text LISP group contained dead code, which suggests that the GIL students had a better understanding of the way in which their programs manipulated data. Dead code is likely to arise when students make stepwise changes in a program in an attempt to get correct output. The text LISP students often had long episodes of successive

Table 1: The GIL Curriculum

1 Introduction to LISP
- 1.1 Programming in LISP
- 1.2 Getting Started: Functions
- 1.3 Atoms and Lists
- 1.4 Balancing Lists
- 1.5 Functions for Operating on Lists
- 1.6 Extracting Information From Lists: *first* and *rest*
- 1.7 Combining Functions

2 Manipulating Lists
- 2.1 Building Lists: *cons*, *list* and *append*
- 2.2 Specifying More Than One Input for a Function
- 2.3 Why Create Functions?
- 2.4 Looking at Lists from the Back End
- 2.5 Using Input Data More Than Once
- 2.6 Making Your Programs General

3 Writing Programs With Variables
- 3.1 Input and Output Data
- 3.2 Building A Program Graph With Variables
- 3.3 Running a Program
- 3.4 Creating Variables
- 3.5 Creating and Editing Your Graph

4 Arithmetic Functions

5 Conditional Processing

6 Predicates

7 Conditionals
- 7.1 The *Cond* Structure
- 7.2 Building a *Cond* Structure on the Computer

8 Logical Functions

changes, many of which were irrelevant but remained in the final solutions. In contrast, the graphical representation in GIL may have helped students discern the structure of their partial solutions and better understand the state of the data at each point in the program, resulting in more focused diagnosis and repair of errors.

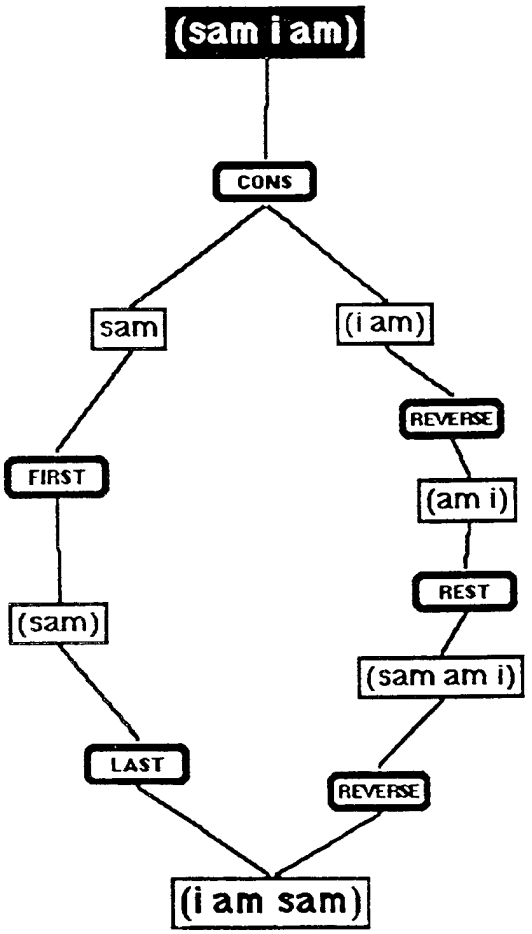
We also assessed when students chose to modify their programs. The text LISP students typically changed their code when they received an error from the interpreter; only once did a text LISP student make a change after simply examining the program. In contrast, all GIL students made such a change at one time or another. This suggests that the GIL students were better able to understand how their programs would behave and thus could locate errors by examining their solutions so far without requiring feedback from the system, whereas text LISP students relied more on external feedback to find out how their programs would run.

In summary, this study showed that GIL offered significant advantages over standard text LISP. The differences between the GIL group and the text LISP group in the number of solution modifications required, the types of modifications made, the presence of dead code, and the timing of solution modifications suggest that the GIL students were able to construct their solutions more easily, and that they achieved a better understanding of the programs they constructed. The experiment supports the claim that a visual representation can facilitate students' reasoning about complex devices by helping them construct more accurate models of how these devices behave.

2.2 Experiment Two: Direction of Reasoning

An important advantage of GIL's design is that it allows students to work on the components of a solution in whatever order they find most natural (Ranney & Reiser, 1989; Reiser et al., 1989). Students can reason either from the inputs toward the goal data (a forward step) or backward from the goal data toward the original inputs. Reiser et al. (1989) found that novice programmers strongly preferred to take forward steps in their solutions. This order of reasoning is opposite the surface form of the solution — the first function encountered in the body of the text form is in fact the last component of the solution assembled by GIL subjects (see Figure 1). This suggests that an advantage of GIL's graphical system may be that it offers students a way to reason in a more natural order than students who are led to assemble a solution in the order in which the components appear in the text form of the solution (Ranney & Reiser, 1989; Reiser et al., 1989).

In a series of experiments in collaboration with Greg Trafton, we are investigating how the direction of reasoning can affect the difficulty of acquiring expertise in a new domain. In a first experiment, we tested this idea directly by manipulating



(a)

```
(defun rotater (lis)
  (cons (first (last lis))
        (reverse (rest (reverse lis)))))
```

(b)

Figure 1: A completed solution in (a) GIL and (b) the corresponding text form

the direction of reasoning allowed (Trafton & Reiser, 1991). Because we were interested primarily in how the direction of reasoning would affect subjects' planning and implementation of programs, we used GIL in model tracing mode in this experiment to minimize the difficulty of learning to use LISP constructs correctly. GIL interrupted students whenever they assembled an illegal programming step, and the tutor helped them modify the step to make it legal. Students were also interrupted when they embarked on a poor strategy, although GIL's assistance at that point was limited to pointing out that the current step was legal but would not help in the problem. Thus, we expect problem difficulty to be primarily a measure of the complexity of planning a solution and repairing the plan if needed. In this experiment, we found that subjects who worked forward solved the problems in less time and with fewer errors than subjects who were required to use backward reasoning. They also deleted fewer of their correct steps, suggesting they were less likely to reach impasses.

The experiment conducted during the first year of the present contract continues this line of work. In this experiment, we investigated two factors that may have affected the results in our previous study. We have argued that reasoning in the forward direction is more congruent with the way novices reason about the behavior of functions in the programming language. We have also argued that this reasoning is facilitated by the intermediate results used in GIL's graphical interface (Reiser, Kimberg, Lovett, & Ranney, 1991). However, one might raise the concern that the advantage of forward over backward reasoning in the domain has been shown only in a modified representation, the graphical representation with intermediate products, rather than in a more traditional text representation. Does enabling forward reasoning for students truly allow them to communicate their solutions in a representation more congruent with their planning, or is the advantage artificially due to some characteristic of the strategies elicited by the presence of intermediate products? We have argued that the intermediate products that form an essential part of GIL's graphical representation provide a more useful model of programming to provide to novices, but as yet there is no empirical evidence that such intermediate reasoning products indeed facilitate reasoning. In order to investigate this issue, we constructed another version of GIL's interface in which students do not enter the input and output data for each function they use. This experiment will test whether the intermediate products facilitate students reasoning, and whether the advantage for forward over backward reasoning artificially arises from strategies elicited when students use these intermediate products.

A second motivation for the experiment concerns the use of GIL's model tracing feedback in our earlier experiment. One might argue that the advantage of forward over backward reasoning arises in some way from a possible bias in GIL's model

tracing feedback. In fact, we rewrote many of GIL's messages for this experiment to be as neutral as possible between describing the behavior of LISP functions in either a forward or backward direction. However, the search space is more constrained when reasoning forward than when reasoning backward in the domain of LISP programming because a function and its input uniquely determine an output result, whereas there may be many different inputs to a function that would yield a particular output (Trafton & Reiser, 1991). Consequently, in some cases the features of an illegal output could be described in error feedback more precisely than examples of an illegal input. Thus, it is possible that the error explanations were more helpful for the forward reasoning subjects. The present experiment also allows us to investigate whether the observed facilitation of forward over backward reasoning was due to GIL's explanations rather than a greater congruence with students' natural reasoning. In this experiment, subjects learn to solve LISP problems without any model tracing feedback. In the exploratory version of GIL, GIL does not interrupt. Instead, students are free to assemble any graph containing illegal steps or poor strategies. Students are able to test their programs to discern whether their program produces the intermediate and final results they have specified, and can run their programs with new data to test their generality (see Merrill, Reiser, Ranney, & Trafton, 1991 for a more detailed description of the exploratory versus model tracing versions of GIL).

Subjects were required to reason in either the forward or the backward direction, and constructed their graphs either using intermediate data nodes or without these data nodes. Thus, there are four conditions in this experiment, all constructed using GIL in its exploratory mode:

1. Forward Reasoning, Intermediate Nodes: Subjects are forced to reason in the forward direction, from the original inputs toward the goal. Each step must use given data as input, and produces new output data (until the final goal output is achieved). Intermediate nodes are required for the input and output of each function used.

2. Backward Reasoning, Intermediate Nodes: Subjects are forced to reason in the backward direction, from the goal data toward the original inputs. Each step uses a function on new inputs to obtain a goal node, setting these new inputs as subsequent goals that must be achieved. Intermediate nodes are required for the input and output of each function used.

3. Forward Reasoning, No Intermediate Nodes: Subjects are forced to reason in the forward direction. No intermediate nodes are used.

4. Backward Reasoning, No Intermediate Nodes: Subjects are forced to reason in the backward direction. No intermediate nodes are used.

Figure 2 displays the form of GIL solutions in the Intermediate and No-Intermediate conditions.

First, we consider whether subjects reasoning forward were able to solve the assigned problems more quickly than subjects reasoning using backward steps. Subjects in the forward intermediate condition required approximately 30% less time to solve the assigned problems than the subjects in the backward intermediate condition. This replicates our earlier result and rules out the possibility that the facilitating effect of forward reasoning arises from an interaction with GIL's explanations. This result supports our claim that forward reasoning is more effective in this domain.

Students who solve problems in the exploratory version of GIL typically require a number of solution attempts before they construct a correct program. Therefore, we can divide subjects' problem solving into two stages: a building stage in which the student constructs the initial graph, and a debugging stage in which the student tests and corrects any errors in the program. The solution time differences were present in both the build time and the debugging time, suggesting that subjects had more difficulty both in planning their original solution for each problem and in debugging any errors that occurred. Interestingly, the differences were present in the Forward versus Backward intermediate conditions, but there were no differences in solution time between the forward no-intermediate condition and the backward no-intermediate condition.

The solution time measure itself cannot be used to discern whether the intermediate nodes facilitated problem solving. The subjects in the two Intermediate conditions had to construct intermediate data nodes in addition to selecting the functions in their program, and these additional interactions of typing or selecting these intermediate nodes add to the time to construct solutions. Therefore, comparisons of solution times between the intermediate and no-intermediate conditions are not informative concerning the problem solving difficulty.

Table 2 presents the number of erroneous solutions constructed during the learning session. These results are consistent with the solution time results. Again, subjects in the Forward Intermediate condition made fewer errors than subjects working Backward. The difference between the Forward and Backward No Intermediate conditions is not reliable. The number of functions deleted or replaced in students' attempts to construct a properly running program is another measure of the relative difficulty subjects experienced in constructing and modifying solutions to solve the assigned problems. These data, shown in Table 3, reveal a similar pattern.

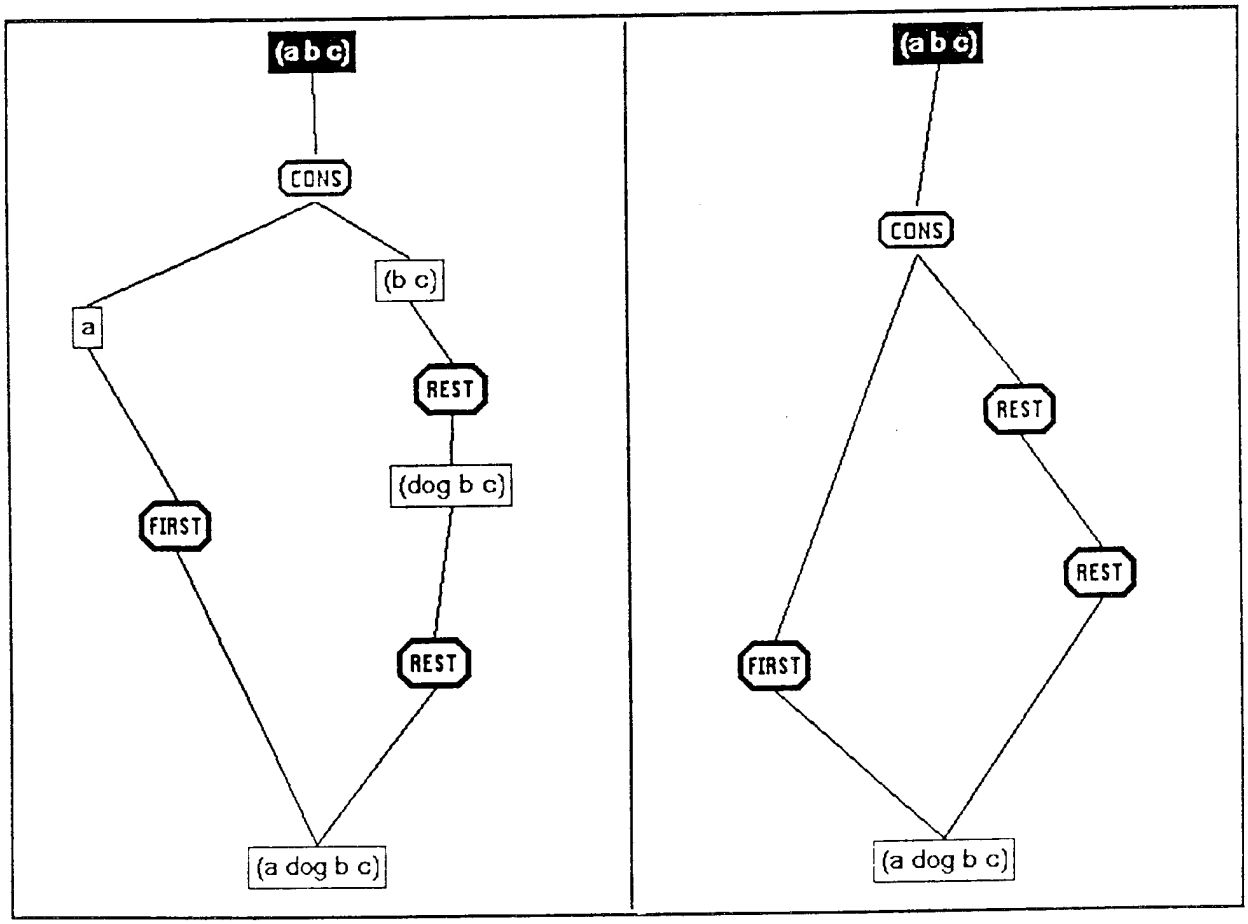


Figure 2: The two types of solutions in Experiment Two. In the solution on the left, the student has specified intermediate products — the input and output for each function selected. The solution on the right is from the version of GIL in which intermediate products are not used.

Table 2: Number of erroneous solutions in the learning session

	Interm Nodes	*No Interm Nodes*
Forward	4.2	10.2
Backward	9.5	11.1

Table 3: Number of functions deleted or replaced

	Interm Nodes	*No Interm Nodes*
Forward	8.3	26.3
Backward	22.6	25.6

Generally, the subjects working forward in the intermediate node condition solved the problems more quickly, made fewer errors, and deleted fewer functions than the subjects working backward. The Forward Intermediate subjects also performed better than both groups of No Intermediate subjects. However, these measures reveal little difference between the forward and backward version of the No-Intermediate conditions.

One interpretation is that the advantage for forward reasoning somehow arises out of a reasoning strategy elicited by the presence of these intermediate nodes. It is important to stress that performance in the No-Intermediate conditions is worse than performance in the Intermediate conditions. In fact, the performance in the No-Intermediate conditions is approximately equal to the performance in the Backward Intermediate condition for most measures. Therefore, although the expected forward backward difference arises only when intermediate nodes are present, this cannot be attributed to an artificial strategy evoked by the presence of intermediate nodes. Instead, it appears that forward reasoning indeed can facilitate reasoning in this domain, but that the advantage only appears when students can use the graphical representation to record their intermediate results.

Another way to directly examine the effects of the intermediate nodes is to compare the correctness of the first solution constructed in each problem. These results, shown in Table 4, reveal that when students were required to enter intermediate nodes, they constructed more accurate solutions than when they built graphs without these intermediate products. These results provide strong support that the

intermediate nodes indeed facilitate the students' reasoning.

Table 4: Correctness of Initial Solution Attempts

	Interm Nodes	*No Interm Nodes*
Forward	92.2%	84.3%
Backward	86.4%	81.2%

We suggest that programming in the forward direction is more congruent with the way students construct their solutions and reason about the behavior of computer programs, so students working forward encounter less difficulty constructing and repairing their solutions. The intermediate products allow students to explicitly test their understanding of the functions they use by allowing them to predict what output a function returns when given a particular input and observe whether their predictions are correct. The combination of working in a direction that is congruent with the manner students reason and using intermediate products to help their understanding of functions greatly facilitates programming skills for the subjects working in the forward intermediate condition.

3 Progress on GIL

We have made progress on our GIL tutoring system on several fronts. First, we are extending the graphical representation to handle more complex programs, so that we can test our principles of designing facilitating visual representations and to empirically evaluate the theory of visual representations with a comprehensive curriculum. Second, we are extending the model tracing capabilities to explore how to encode causal knowledge in a problem solving system, and to use this knowledge as the basis for tutorial explanations.

3.1 The Macintosh version of GIL

During this year we have completed the port of our GIL tutoring system to the Macintosh. Our system was originally written in Interlisp and the LOOPS object system on Xerox LISP Workstations. Now that Macintosh computers have become powerful enough to run LISP programs at reasonable speed, they are a better computer on which to develop the software and conduct our experiments. Macintosh computers are more reasonable machines for the delivery of instructional software

in training centers and schools than Xerox workstations. Furthermore, although several years ago Xerox workstations were the ideal computer on which to develop intelligent tutoring systems because of their advanced programming environment, this hardware is now outdated and difficult to maintain. Fortunately, many of the features of the LISP programming environment from the Xerox workstation are now available in the Macintosh LISP environment.

The staff working on the port during this year consisted of Eliot Handelman (post-doctoral research associate) and Adnan Hamid (undergraduate student research assistant).

There were two components of the port of GIL to the Macintosh: the LOOPS object system and the interface support for GIL. Our project makes extensive use of the LOOPS object-oriented programming system. GIL's knowledge base of problem solving rules and plans are organized in LOOPS class hierarchies, and class hierarchies are essential for organizing the various types of data structures used by the interface. In addition, the inheritance properties of this hierarchy are used in the interface to conditionalize various aspects of the behavior of the interface on experimental conditions. Hence, we first wrote an implementation of LOOPS in Common LISP. Thus, with this LOOPS environment, much of our code from the Xeroxes can be run without modification on the Macintosh.

The second major component of the port was converting from Interlisp to Common LISP. A good portion of the conversion was done automatically by programs available from other researchers written to translate between various LISP dialects. Much of the interface code, however, needed to be rewritten using the Macintosh Common LISP graphical routines. Again, rather than rewriting the Interlisp interface code itself, we chose to implement support for the Interlisp window, bitmap, and mouse-drive event handling routines within the Macintosh Common LISP software. This greatly simplified the porting task.

The port was completed in July 1991. All subsequent development of GIL is now taking place on Macintosh computers. The net result is that our system runs more quickly, and now runs on machines that are easier to maintain and much more commonly available. A Macintosh sufficiently powerful to run GIL currently costs in the $4,000–$6,000 range (university prices) — a Macintosh II (IIsi, IIci, or IIfx) with a 2-page display.

3.2 The GIL Model Tracer and Explainer

We are examining data from students who have used GIL in its exploratory mode to characterize the types of conceptual difficulties students encounter in the more advanced curriculum. In this first year of the contract, we have extended the diagnostic

capabilities of our plan-based model tracing system to produce better explanations upon student errors. An example of an explanation in response to a student's error is shown in Figure 3. The model tracer has discerned that the variable used by the student is incorrect in this context. If the student requests more information, GIL provides the second level of help, shown in Figure 4. In this additional hint, GIL suggests what variable to use instead, and reminds the student of the roles of these two variables. Most of GIL's explanations focus first on flagging the error situation, and then reminding the student of the relevant goals.

Building the model tracer for sections the lessons on conditionals (Sections 5–8 in Table 1) has provided a good test of our plan-based model tracer. During our work on the model tracer this year, we have evaluated how well this scheme has worked in comparison to the production rule model tracer we used in our earlier versions of GIL. There are several important advantages to this plan-based model tracer.

Many of the advantages of the plan-based approach arise from the difficulty in providing students freedom to compose solutions in the manner they desire. This poses problems for production rule approaches because writing rules for many such orders creates much duplication in the production set. This makes it very costly to construct and debug a production rule set. These problems can be circumvented by constraining the order in which students can compose their solutions. For example, in the CMU LISP Intelligent Tutoring System (Anderson et al., 1989; Reiser et al., 1985), students are constrained to enter their solutions in top-down left-to-right order. That is, they cannot code embedded parts of the program without coding the outside function calls. Recent developments in *compiling* the various paths of the production rule system through the problem space have greatly added to the efficiency and flexibility of production rule model tracing systems (Corbett, Anderson, & Patterson, 1990). However, such systems still require students to use placeholders for missing code if they want to construct their solutions in a different order from the final surface form of the code.

The plan-based model tracing system in GIL provides students freedom to construct their solutions in any order desired. The plan-based problem solver expands the entire goal-plan tree of solution decompositions. Thus, the model tracer can can search the entire tree for any step the student enters. As a consequence, students can enter their solutions in a more flexible order. In conditional problems, such as in Figure 3, students can work on any test or action at any point, and can construct embedded function calls within a test or action in any order. This flexibility also relies upon GIL's graphical representation, which provides a natural medium to construct subparts of solutions and then link them together when the various parts have been completed.

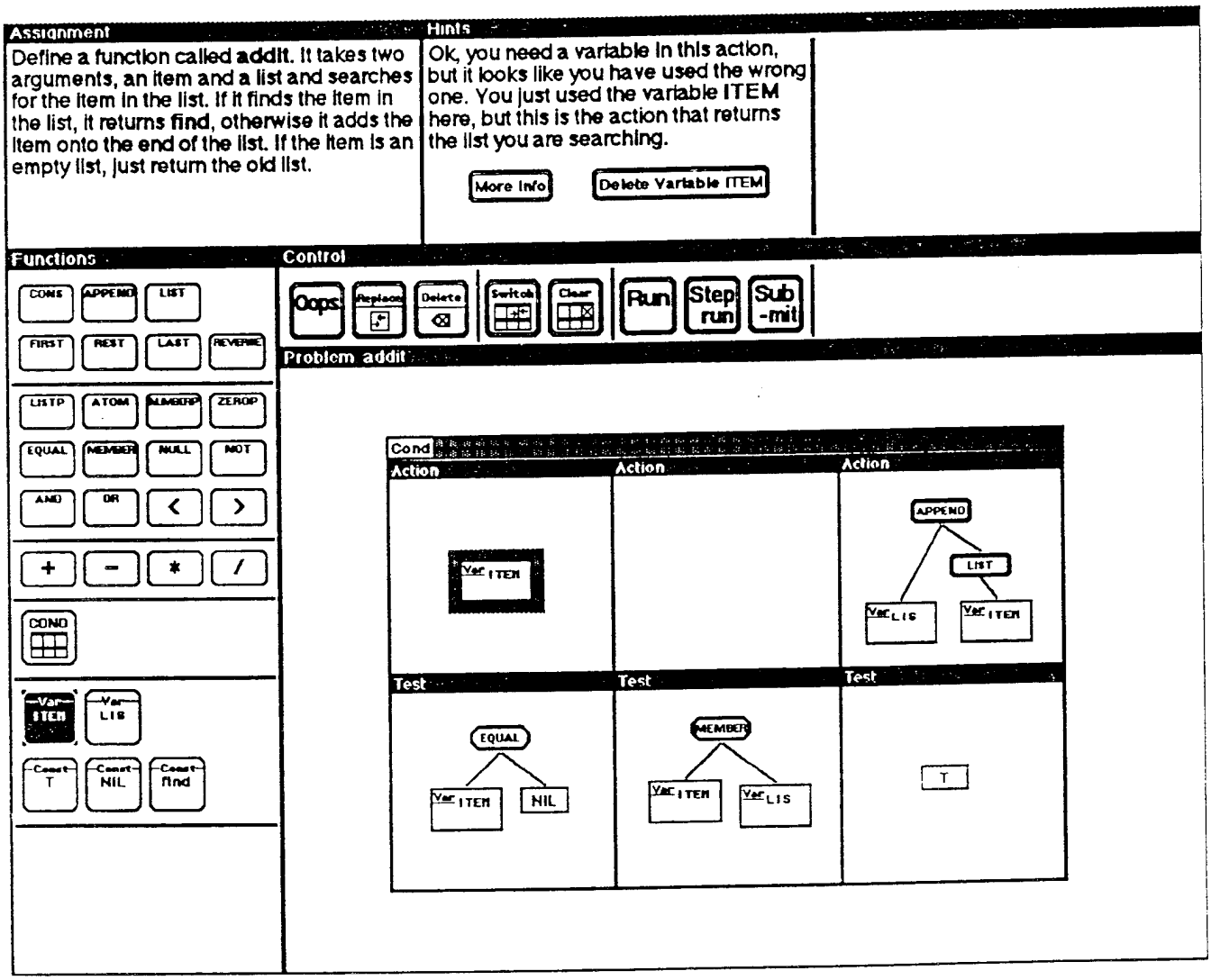


Figure 3: GIL's response to a solution in which the student has used the wrong variable.

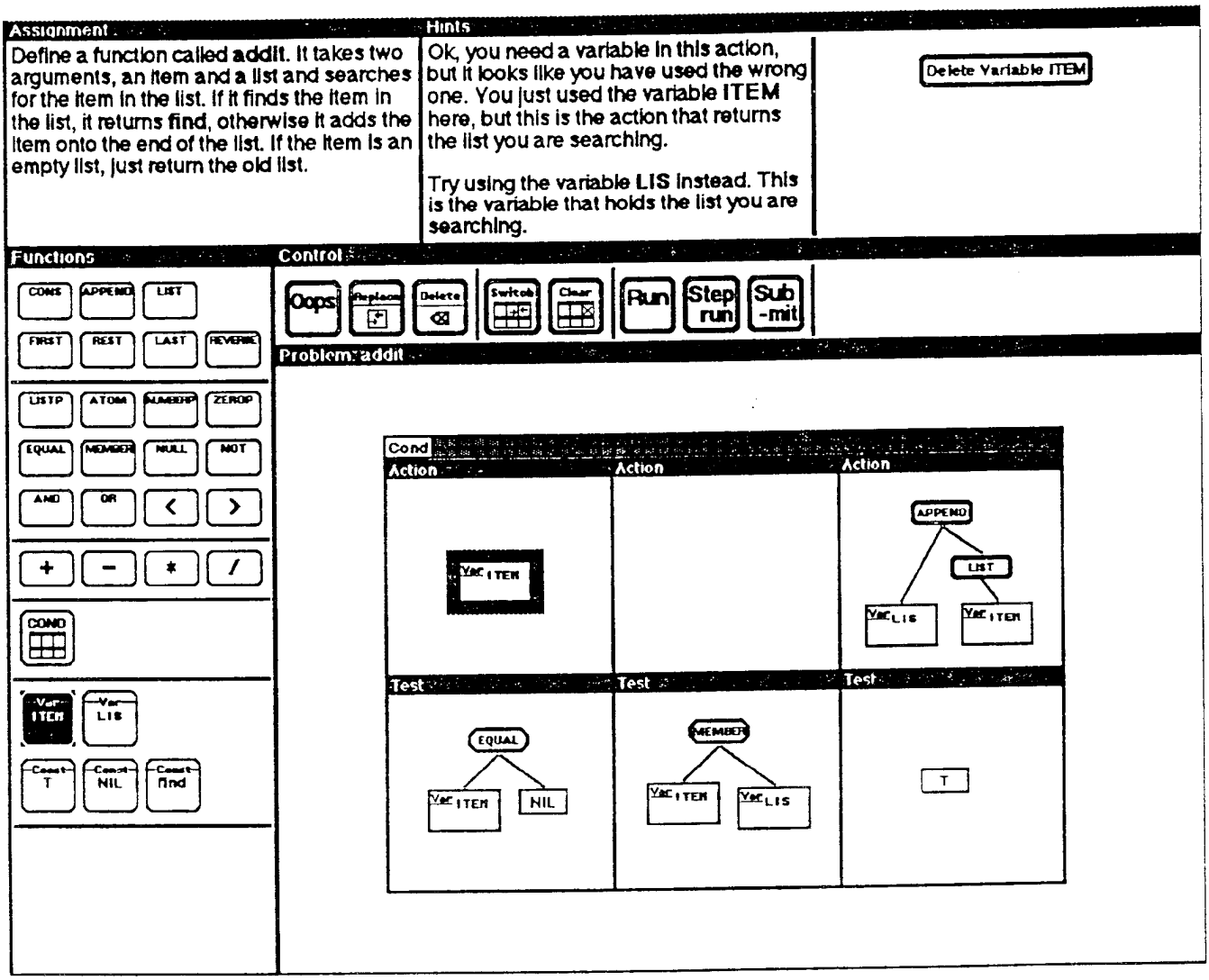


Figure 4: GIL's second level of help, which provides a particular suggestion about how to modify the student's current step in the solution.

Second, a difficult problem in a production rule model tracer is ambiguity. In many types of problems, a single step may not disambiguate between a number of plans, because that step may be shared by those plans. Thus, earlier production rule model tracing systems typically contained rules that represent steps done for multiple reasons. This greatly complicates the task of writing a production rule set to cover all possible solution paths desired. Another possible compromise is to have the students disambiguate their intentions by responding to queries, such as which case of the conditional they want to code next, but these types of interventions may interfere with the students' planning.

In contrast, the plan-based model tracer is designed to handle ambiguity. When a step fits multiple plans, the model tracer simply keeps track of all possible nodes in the goal-plan tree that a particular step may implement. With each subsequent step, the model tracer checks which of these possibilities are consistent with the newly entered step and prunes the possibilities accordingly.

Another advantage of the plan-based model tracing system is that it allows the tutor to respond to errors in the context of the students' plan. This can be done to some extent in production rule model tracing systems, because the production system may encode multiple rules for a particular step that discriminate between different plans in which that step may take place. However, encoding the plans explicitly provides a more powerful and consistent method for providing feedback on the plan in which a step occurs. Furthermore, the plan-based methodology provides a more robust system to respond to student errors. A production system model tracer can compare a students' current step to the set of rules considered by the problem solver. In cases where the student has a correct intention but implements it incorrectly, such a model tracer can provide a helpful response. However, a number of important errors arise when students skip necessary steps or attempt to implement the subgoals of a plan in an incorrect order. Knowledge of the students' plan enables the model tracer to recognize that the student has taken a step which will later be necessary but is missing some intermediate steps. Such an analysis is difficult for a rule-based model tracer, which would need to consider steps several cycles "ahead."

Figure 5 shows an explanation in GIL of one such situation. Here the student has begun building the action component of the third case of the conditional. The student has begun by selecting the *append* function, and then brought in the variable *lis* and linked it as input to the *append.* Then, the student placed the variable *item* in the action, and linked it as the second input to the *append.* At this point, GIL intervenes. It recognizes, like a rule-based system would, that the second input to the *append* is not this variable, but should be a function call. However, by matching the students' solution so far against possible plans for implementing this program, GIL can also recognize that the variable should indeed be linked through a path to

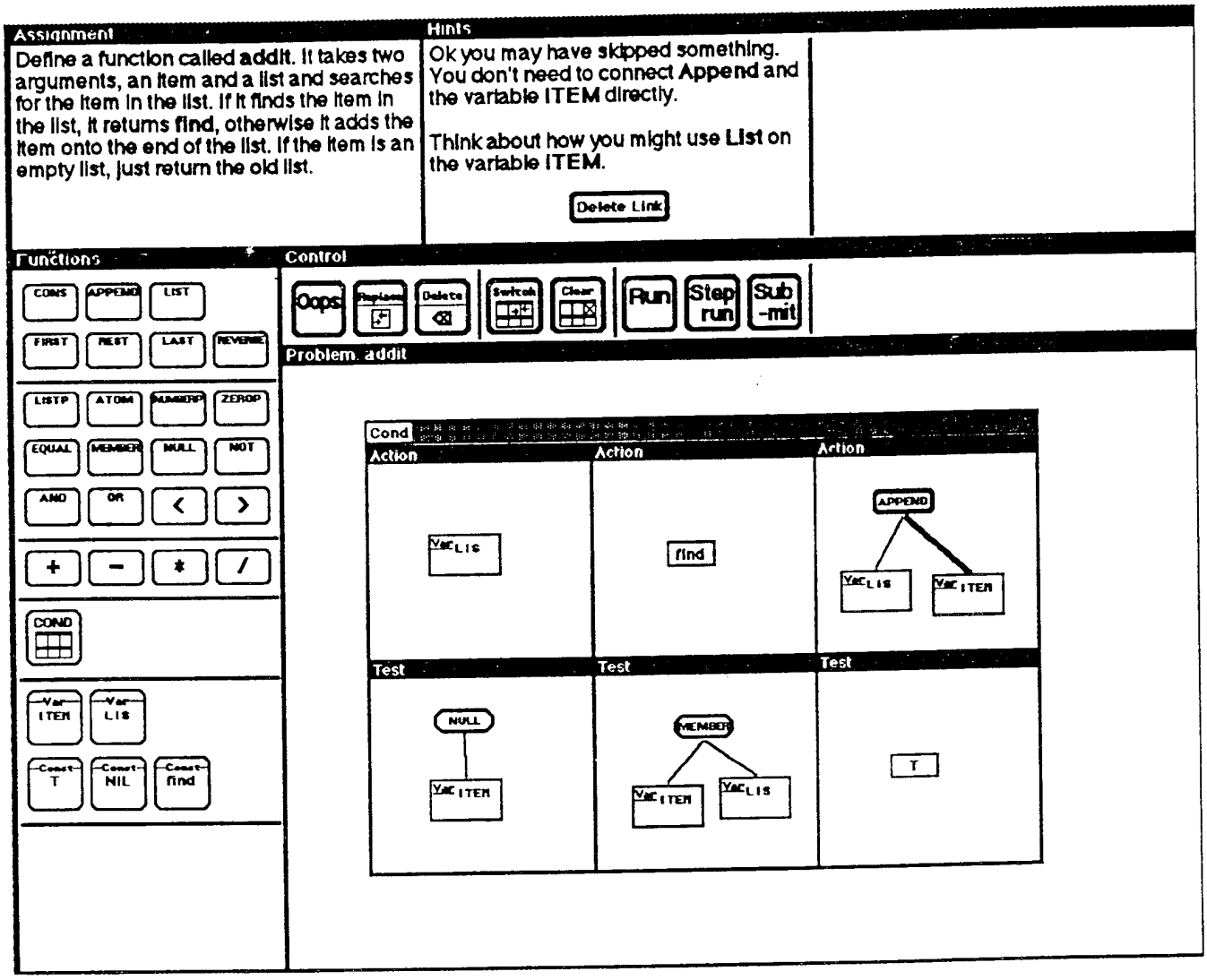


Figure 5: Plan-based analysis in GIL.

the function, and it can suggest an additional operation that should be performed on the variable before it is connected as input to the *append*.

Another major issue we are currently exploring in our work on the model tracer is the potential for placing more of GIL's feedback under student control. The lessons on conditionals (Sections 5–8 in Table 1) provides an excellent opportunity to explore this issue. A number of important types of misconceptions that occur when students master this material concern strategies for ordering the clauses in a conditional expression. Like other model tracing tutors (e.g., Reiser, Anderson, & Farrell, 1985; Anderson, Conrad, & Corbett, 1989), the current version of GIL interrupts the student when a case is first begun that it recognizes to be in the wrong order.

Our current strategy is to provide the student with an opportunity to keep working on getting each of the individual cases correct and fix the ordering problem later. Our reasoning is that it may be useful to allow the students to correctly handle the individual subgoals before attending to this higher level repair. An example of this type of feedback is shown in Figure 6. Here the student has performed part of the planning correctly. The student has understood two of the important subgoals of this problem, testing for membership of the target item in the given list and guarding against a *null* target, and has made good progress in correctly achieving each of these subgoals in the first and second clauses of the conditional. The student has not correctly understood (or has not yet attended to) the additional subgoal of ordering the tests correctly so that each type of data falls into the appropriate test. The tutor points out this problem with the student's solution so far, and lets the student choose between receiving a suggestion about how to repair the difficulty or completing the particular subgoals and then repairing the error later. This strategy has the advantage over an intervention that forces the student to repair the order immediately, because it allows the student to continue working on the current subgoal before attending to the ordering of cases problem, and it allows the student to complete each subgoal before reasoning about how to combine those subgoals. However, it has the disadvantage that it does interrupt the student and potentially distracts from the current problem solving.

In the second year of this contract, we plan to investigate a strategy whereby more global problems such as misordered or missing cases are pointed out only when the student requests a hint or submits the solution, rather than having GIL interrupt as soon as it diagnoses that such an error has occurred. This strategy would have several advantages. First, any explanation of why the cases belong in another order would be more sensible when the student was further along in the solution, having specified the contents of the two misordered cases. Furthermore, it may be less profitable to interrupt the student's local problem solving within

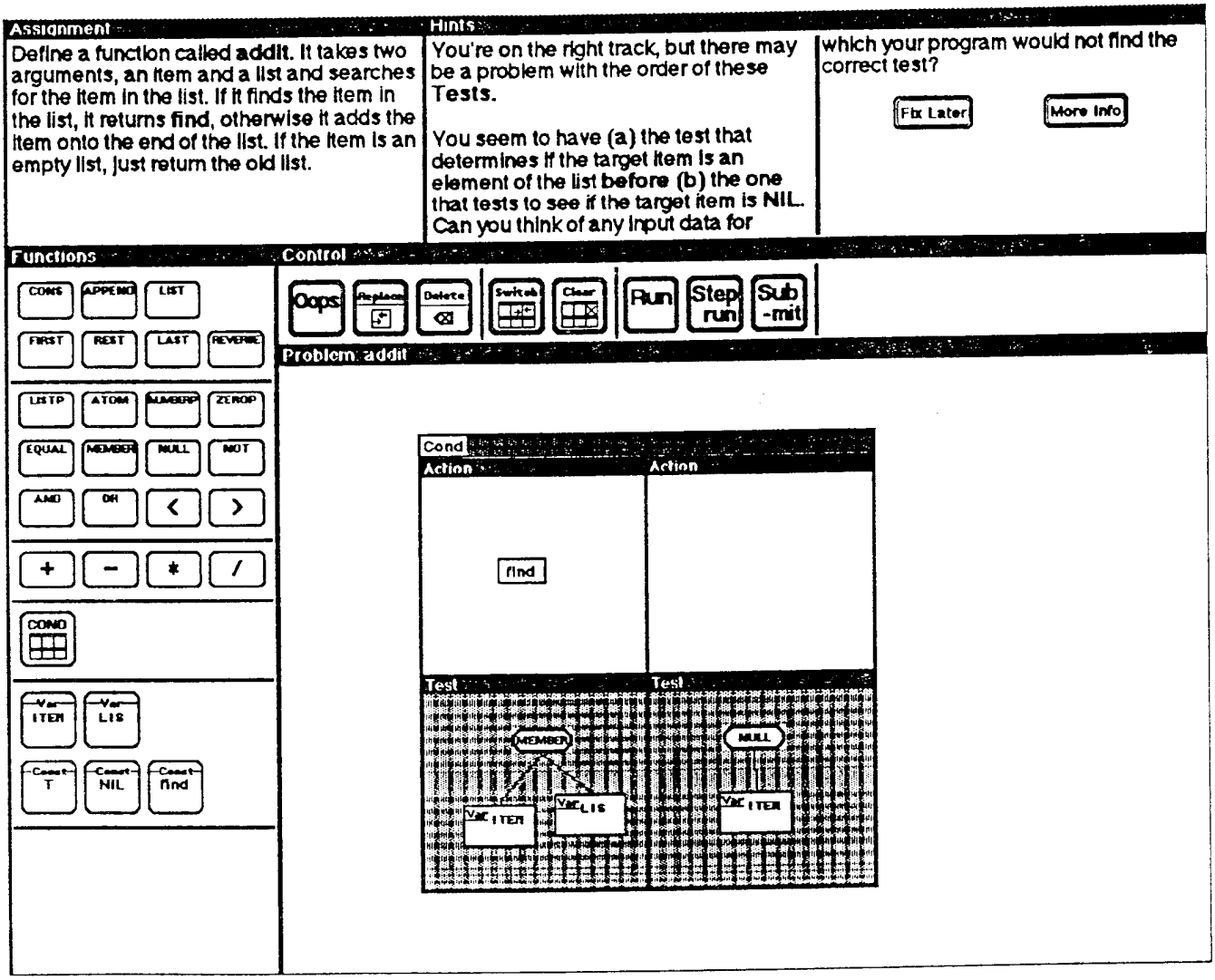


Figure 6: The student has made progress toward correctly achieving the major subgoals, but has misordered the components of the solution.

the case, rather than waiting until the two cases are completed. In general, we are attempting to augment GIL's model tracer so that it can discriminate between those errors students should repair immediately, because they will interfere with further problem solving, and those errors that can more easily be understood and repaired when more of the solution has been corrected. This is an important characteristic of the advice offered by the human tutors we have studied (Merrill et al., 1991).

3.3 The Graphical Representation

During the first year of this contract we have designed the extension of the GIL curriculum to the topic of iteration. We have begun the implementation of a prototype for the interface for these lesson. The initial lesson on iteration includes graphical constructs for list iteration (the common lisp construct *dolist*) and numeric iteration (*dotimes*). In addition, the prototype includes the constructs *let* and *setq* to create and bind local variables to values, which are needed for most iterative programs. In the next year, we plan to complete the prototype and pilot test it with students.

3.4 GILT: The GIL Text-Based Program Editor

We are beginning to to attempt to delineate the precise benefits of graphical interfaces and to determine which, if any, of these benefits can be embedded in a textual interface. To investigate these issues, we have begun development of a structured editor called GILT (*Graphical Instruction in LISP Text-version*), which possesses all of GIL's editing and testing facilities, but does not use the graphical representation of LISP programs. Instead students build the traditional text form of a LISP program, (as shown in Figure 1b). GILT is being designed to capture all beneficial aspects of GIL that are not solely due to the graphical LISP version.

We are constructing GILT in order to be able to systematically explore the benefits of the graphical representation used in GIL. GILT will provide a control condition, in which much of the syntactic and interface difficulties of learning a programming language are minimized. If students learn programming more easily with GIL, the benefits can be attributed to the graphical representation employed in GIL. In addition, GILT will enable systematic exploration of particular aspects of the graphical representation by varying which capabilities are included, such as access to intermediate results, explicit animation of the flow of control during the execution of a program, and so on.

A second motivation for the construction of GILT concerns assessing what students have learned when they have learned to program with GIL. We believe that GIL students are learning the same programming concepts as students who learn

the traditional text-based representation, and GIL helps students master these concepts more easily and quickly. However, the possibility exists that the benefits GIL students receive are limited to the graphical form of the language. Therefore, to address this issue, we have planned a series of transfer studies, in which GIL students will first solve problems in the graphical representation, and then transfer to the text LISP reprenstation, using GILT. To test this, we will compare the performance of students who solve problems in GIL and then translate them into GILT to students who solve them only in GILT. We expect students who learn using GIL's graphical representation will master the material more quickly and to a greater level of proficiency than students who use GILT only.

The staff working on the design and implementation of GILT during this year consisted Adnan Hamid (undergraduate student research assistant) and Douglas Merrill (graduate student).

A student interaction with GILT will be much like one with GIL. The GILT student will select a function name from a function menu always present on the left side of the screen, and will then select a location to place the selected function. The function will appear on the screen not as an icon (as in GIL), but rather as traditional text LISP, complete with parentheses. After placing the function, the student will be asked to put a right parenthesis in the appropriate place, thereby telling GILT what the arguments to the function are. In this way, students will build traditional text LISP definitions, but GILT will avoid many types of low-level syntactic difficulties, such as matching parentheses.

The students will be able to execute their programs on input data of their choice and see the resulting output. They will also be able to examine fine-grained intermediate results, although they will not be asked to provide them (at least in the early stages of GILT research). The students will be able to replace, delete, and cut and paste functions, to maximize their ability to build the solution in any order necessary.

During this year, we have completed a prototype of GILT with much of the editing functionality. In the coming year, we plan to complete the editing capabilities, add a facility for running programs, and then test the pilot version with students. Based upon the pilot results, we will likely fine-tune the interface for usability, and then perform our studies on the effectiveness of graphical representations by comparing students learning with GIL and GILT.

4 Publications and Presentations of the Research

4.1 Colloquia

Colloquia describing this research were presented by the Principal Investigator at the University of Michigan, Columbia University Teachers College, and Northwestern University.

4.2 Conference Presentations and Publications

Bauer, M. I., & Reiser, B. J. (1990). Incremental envisioning: The flexible use of multiple representations in complex problem solving. *The Proceedings of the Twelfth Annual Conference of the Cognitive Science Society* (pp. 317–324). Hillsdale, NJ: Erlbaum.

Bauer, M. I., & Reiser, B. J. (1990). The flexible use of multiple representations in AI problem solvers. *The Proceedings of the 1990 AAAI Symposium on Model-Based Reasoning.*

Merrill, D. C., Reiser, B. J., Ranney, M., & Trafton, J. G. (1991). *Effective pedagogical techniques in human tutors and intelligent tutoring systems.* Submitted for publication. Also released as Technical Report #51, Cognitive Science Laboratory, Princeton University.

Reiser, B. J., Beekelaar, R., Tyle, A., & Merrill, D. (1991). GIL: Scaffolding learning to program with reasoning-congruent representations. *The Proceedings of the International Conference on the Learning Sciences* (pp. 382–388). Association for the Advancement of Computing in Education.

Reiser, B. J., Kimberg, D. Y., Lovett, M. C., & Ranney, M. (1991). Knowledge representation and explanation in GIL, an intelligent tutor for programming. In J. H. Larkin & R. W. Chabay (Eds.), *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches.* Hillsdale, NJ: Erlbaum.

Trafton, J. G., & Reiser, B. J. (1991). Providing natural representations to facilitate novices' understanding in a new domain: Forward and backward reasoning in programming. *The Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society* (pp. 923-927). Hillsdale, NJ: Erlbaum.

5 References

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science*, 13, 467–505.

Corbett, A. T., Anderson, J. R., & Patterson, E. G. (1990). Student modeling and tutoring flexibility in the Lisp Intelligent Tutoring System. In C. Frasson & G. Gauthier (Eds.), *Intelligent tutoring systems: At the crossroads of artificial intelligence and education* (pp. 83–106). Norwood, NJ: Ablex Publishing Corporation.

Merrill, D. C., Reiser, B. J., Ranney, M., & Trafton, J. G. (1991). *Effective pedagogical techniques in human tutors and intelligent tutoring systems.* Technical Report No. 51, Cognitive Science Laboratory, Princeton University, Princeton, NJ.

Ranney, M. & Reiser, B. J. (1989). Reasoning and explanation in an intelligent tutoring system for programming. In G. Salvendy & M. J. Smith (Eds.), *Designing and using human-computer interfaces and knowledge based systems: Proceedings of the Third International Conference on Human-Computer Interaction* (pp. 88–95). New York: Elsevier Science Publishers.

Reiser, B. J., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for LISP programming. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* Los Angeles, CA.

Reiser, B. J., Beekelaar, R., Tyle, A., & Merrill, D. C. (1991a). GIL: Scaffolding learning to program with reasoning-congruent representations. In *The International Conference of the Learning Sciences: Proceedings of the 1991 conference* (pp. 382–388). Evanston, IL: Association for the Advancement of Computing in Education.

Reiser, B. J., Kimberg, D. Y., Lovett, M. C., & Ranney, M. (1991b). Knowledge representation and explanation in GIL, an intelligent tutor for programming. In J. H. Larkin & R. W. Chabay (Eds.), *Computer assisted instruction and intelligent tutoring systems: Shared issues and complementary approaches.* Hillsdale, NJ: Erlbaum.

Reiser, B. J., Ranney, M., Lovett, M. C., & Kimberg, D. Y. (1989). Facilitating students' reasoning with causal explanations and visual representations. In D. Bierman, J. Breuker, & J. Sandberg (Eds.), *Proceedings of the Fourth International Conference on Artificial Intelligence and Education* (pp. 228–235). Springfield, VA: IOS.

Trafton, J. G. & Reiser, B. J. (1991). Providing natural representations to facilitate novices' understanding in a new domain: Forward and backward reasoning in programming. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society* (pp. 923–927). Chicago, IL.